

InterSymp-2007, Baden-Baden, Germany, July 30 to August 4, 2007.

# Enhancing the *Face* of Service-Oriented Capabilities

By

Kym J. Pohl

CDM Technologies Inc.

[kpohl@cdmtech.com](mailto:kpohl@cdmtech.com)

## **Abstract**

*With today's focus toward discoverable web services, Service-Oriented Architectures (SOA) are becoming increasingly prevalent. To support an effective interaction between services and their clientele, the sophistication of the interface, or face, such services present is of critical importance. Without a rich, expressive nature, such services struggle to satisfy the industry promises of reuse, composability, and reduced inter-component dependencies. Especially relevant for domain-oriented applications, services must present sufficient levels of expression to allow for an effective exchange of relevant context. Further, such communication should be offered in an asynchronous manner to promote both work flow efficiency and limited coupling.*

*This paper discusses several concepts and technologies that can significantly enhance the effectiveness of SOA-based capabilities. Technologies including JavaBeans, embedded property change management, and Object/Relational Mapping are leveraged to produce a client interface architecture rich in expressiveness, asynchronous efficiency, and industry standards*

**KEYWORDS:** Service-Oriented Architecture (SOA), JavaBeans, property change management, object/relational mapping

## **Introduction**

With the advent of web services [1, 2], the concept of a Service-Oriented Architecture (SOA) has received considerable attention. Apart from offering benefits ranging from component reuse to runtime composition of capabilities, SOAs are laying the groundwork for dynamic semantic discovery. Considerably more powerful than technologies only able to convey a capability's structure (e.g., XML), the technologies associated with SOA endeavor to support the semantic discovery of a particular service's very nature (i.e., domain of service, semantic expectations and implications, etc.). A critical element in successfully attaining this goal is the sophistication of the interface services present to their clients. This paper describes a combination of design concepts and technologies that can be exploited to produce the type of expressive client interfaces indicative of SOA.

## **JavaBeans Component Architecture Supplemented With the *Property Change* Observation Model**

The JavaBeans technology is one of the fundamental elements comprising the Java Component Architecture. JavaBeans, or simply *beans* for short, are essentially blueprinted objects exhibiting the following characteristics:

- Contain properties whose access is provided through standardized accessor methods (i.e., *getter* and *setter* methods).
- Serializable (useful for both persisting and streaming)
- May be enhanced with additional, application-specific methods for providing specific functionality to its users.
- Support *asynchronous* interaction via the firing of *property change* events (i.e., responses can be triggered by notification of the occurrence of previously *listened to* events)
- Can contain associated metadata (i.e., *BeanInfo*) offering a more precise description of its mechanics than can be provided through more simplistic mechanisms (i.e., Java Reflection)

Although a frequently employed technology in mainstream software development, the standardized protocols and embeddable event behavior promoted by the *beans* pattern provides a solid foundation upon which expressive, self-describing and notifying interfaces can be built. Indeed, it is the latter of these capabilities (i.e., property change management) that provides the asynchronous interaction model that enables the successful exploitation of parallel processing environments. As such, property change management warrants a brief discussion as to how this technology can be employed along with the distribution benefits it offers to SOA-based systems.

### **Property Change Management (*Observation Pattern*)**

Complimenting the JavaBeans Component pattern, the property change observation model establishes an implementable pattern allowing beans to essentially *observe* changes in the *bound* properties of other beans. Observing objects are required to implement specialized interfaces that are automatically invoked whenever the particular condition occurs. Although fundamentally scoped to changes in individual properties, this pattern can be extended to support multi-faceted events occurring across heterogeneous sets of beans. Further, although this mechanism operates locally within a single Java Virtual Machine (VM), such functionality can be extended to seamlessly operate across any number of networked VMs through incorporation of an appropriate transport mechanism (e.g., JMS, CORBA, JDO, etc.) together with a degree of distribution management logic. Regardless of whether operating on a purely local basis or operating in a distributed fashion across a network, users of such *publish/subscribe* facilities operate against the exact same interface, transparent to whether this facility was housed locally or across the network.

The JavaBeans component pattern combined with property change management has become an industry standard. Together, these two technologies are an effective mechanism supporting the asynchronous, event-driven interaction model inherent in parallel processing paradigms (i.e., clients are free to perform other tasks while their requests are being processed). Further, this asynchronous model assists in promoting loosely coupled architectures where component interactions occur as initiations of events paired with any number of anonymous reactions. It should be noted that the latter of these features also aligns well with the loosely coupled and extensible architectures of Aspect-Oriented programming.

### **Domain-Centricity**

Some capabilities are inherently domain-centric. That is, a domain-centric capability is one that operates over an expressive model closely representing concepts and entities conceivable in some reality. Such subject matter may be tangible or intangible, represent actual reality or some hypothetical variation.

Regardless, however, such descriptions comprise the domain over which the capability operates. For example, a capability for planning the delivery of goods will quite plausibly operate over certain notions fundamental to the domain of logistics support (e.g., requirements, transports, delivery routes, goods, time schedules, known impediments, etc.) In essence, these notions form the subject matter upon which the capability operates.

In cases where interaction with a capability's clientele makes significant references to such notions and entities, it is useful to structure the client interface around expressive, domain-specific object models, also known as ontologies (Figure 1). Defining an interface explicitly in terms of the objectified *context* that is conveyed between parties offers both an expressive and more natural language by which services and clients can interact. Such enriched discourse can be substantially more effective and natural than the more traditional approach where such domain-specific context is parameterized into a limited set of invocable functions. Further, defining an interface in terms of such objectified, domain-specific context promotes the efficiency and decoupling offered by asynchronous, event-based interaction as well as the architectural simplicity and elegance incurred with an interaction model comprised of basic, object-level operations (i.e., object creation, manipulation, etc.). Further, combining this concept of domain centrality with the complimentary *bean* and *observer* patterns described above yields a standardized, yet expressive, client interface that supports a decoupled, asynchronous interaction model.

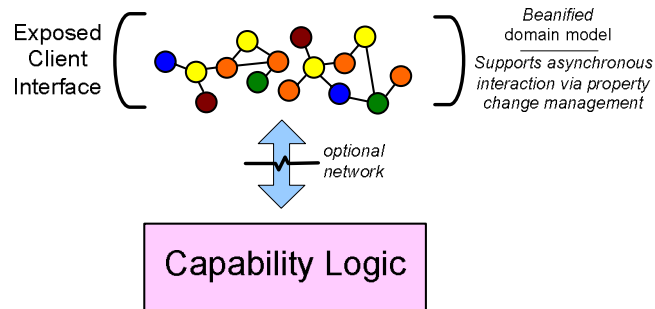


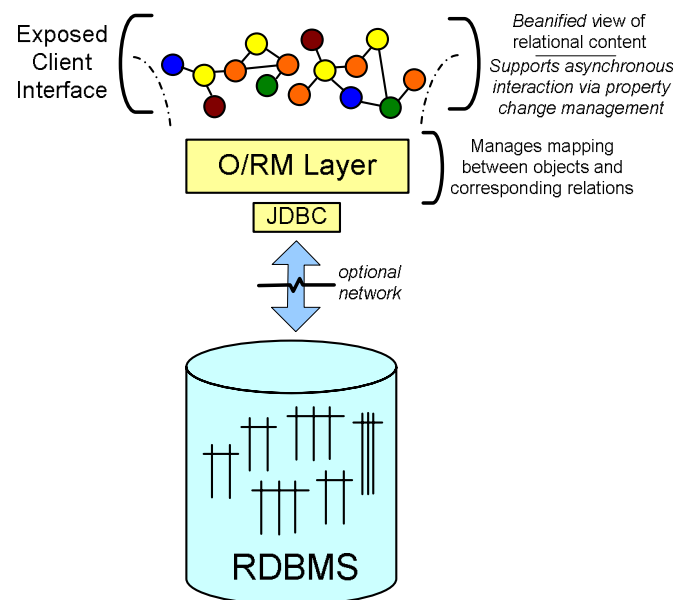
Figure 1 - Domain-Centric Client Interface

To illustrate how these complimentary patterns function together, consider the *Delivery* service briefly described above. The client interface offered by such a service could be composed of an expressive domain model that includes explicit object –level descriptions representing *Requirements*, *Constraints*, *Resources*, etc. Populated by the requesting client, this inter-related cluster of ontology objects can be used to effectively convey the particular problem definition the service is being engaged to solve. Creation and population of instances of these objects would, in turn, trigger the *observing* Delivery service to analyze this information in conjunction with its knowledge of the environment within which the solution would execute (i.e., weather, traffic, security risk, etc.) the deliveries are to be executed within. Because the interaction model is event-driven, the requesting client is free to perform other tasks as its request is being processed. Upon formulating a suitable solution, the service would follow the same asynchronous interaction model as before producing a populated ontology fragment describing the proposed solution (e.g., delivery trips comprised of aspects including timing, stowing, sequencing, routing, mitigation instructions for possible impediments, etc.) To receive such results, clients would simply *observe* the creation/modification of such model fragments relating to their original problem context.

The example above illustrates the use of the property change observation mechanism of the JavaBeans Component Architecture, including the *asynchronous* interaction model that it promotes. The example also highlights the structuring of a client interface around the domain-specific notions that are the fundamental basis of a contextual discourse. Further, such domain-centric interface also provides an effective means of decoupling clients from a capability's underlying architecture (i.e., functional libraries, information management infrastructure, middleware used to distribute the service across a network, etc.).

## Objectifying Relational Schemas

The client interface architecture described above can also be applied to capabilities whose domain models are internally housed within relational environments (e.g., RDBMS, etc.). Whether fully formed services, or simply managed sources of content (i.e., database), such components can expose their domains equally well as collections of interrelated domain objects (Figure 2). Such objectification can be achieved through the application of any number of Object/Relational Mapping (O/RM) technologies (i.e., Hibernate, TopLink, EJB3, JDO, etc.) aimed at addressing the object/relational impedance mismatch [7]. These technologies offer the ability to expose a relational model in an object-oriented form, complete with support for potentially extensive mappings between object fields and corresponding table columns.



**Figure 2 - Presenting an Objectified View of Relational Content**

A key part of O/RM technologies is the specification (typically in the form of *metadata*) and subsequent runtime management of mappings that effectively tie both diverse worlds together (i.e., object classes and relational tables). Finally, by applying the complimentary JavaBeans and property change management technologies, clients to such relational environments can be presented with domain-centric interfaces supporting an efficient, asynchronous interaction model. As such, relationally-oriented capabilities can also enjoy the contextual, efficiency, and decoupling benefits afforded by presenting clients with a domain-centric, event-driven interface.

## Conclusion

With the increased application of Service-Oriented Architecture, there is a growing need to address the ease and efficiency by which such services are employed. This need is even greater when domain-centric capabilities are considered. Technologies that promote standardized, self-descriptive, expressive, and efficient interaction are paramount in supporting the collaboration-intense nature of an evolving, domain-centric and dynamically discovering semantic web topology [1, 2].

## References

- [1] Antoniou G and F. Van Harmelen, “A Semantic Web Primer”, MIT Press, Cambridge, Massachusetts, 2004
- [2] Burke B., R. Richard Monson-Haefel, “Enterprise JavaBeans 3.0 (5<sup>th</sup> Edition)”,
- [3] Daconta M., L. Obrst and K. Smith, “The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management”, Wiley, Indianapolis, IN., 2003
- [4] Erl T., “Service-Oriented Architecture (SOA: Concepts, Technology, and Design”, Prentice Hall Service-Oriented Computing Series, Prentice Hall, Englewood, NJ, 2005.
- [5] Fowler, M., “Analysis Patterns: Reusable Object Models”, Addison-Wesley, Reading, Massachusetts, 1997.
- [6] Fowler M., D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford, “Patterns of Enterprise Application Architecture”, Addison-Wesley, Reading, Massachusetts, 2003
- [7] Hay D., “Data Modeling Patterns: Conventions of Thought”, Dorset House Publishing, New York, NY., 1996.
- [8] Karsai G., “Design Tool Integration: An Exercise in Semantic Interoperability”, Proceedings of the IEEE Engineering of Computer Based Systems, Edinburgh, UK, March, 2000
- [9] Pohl J., “Information-Centric Decision-Support Systems: A Blueprint for Interoperability”, Office of Naval Research (ONR) Workshop hosted by the CAD Research Center in Quantico, VA, June 5-7, 2001
- [10] Pohl J, A Chapman, K Pohl, J Primrose and A Wozniak, “Decision-Support Systems: Notions, Prototypes, and In-Use Applications”, Technical Report, CADRU-11-97, CAD Research Center, Design Institute, College of Architecture and Environmental Design, Cal Poly, San Luis Obispo, CA, January, 1997.
- [11] Rodrigues L., “The Awesome Power Of Java Beans”, Manning Publication, Greenwich, CT., 1998.